

**UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR**



Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

**Herramienta de diseño de juegos tipo mazmorra
para Gamemaker Studio 2**

Autor: Javier Gómez

Tutor: Carlos Aguirre

junio 2019

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución comunicación pública y transformación de esta obra sin contar con la autorización de los titulares de la propiedad intelectual.

La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. del Código Penal*).

DERECHOS RESERVADOS

© 3 de Noviembre de 2017 por UNIVERSIDAD AUTÓNOMA DE MADRID

Francisco Tomás y Valiente, nº 1

Madrid, 28049

Spain

Javier Gómez

Herramienta de diseño de juegos tipo mazmorra para Gamemaker Studio 2

Javier Gómez

C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A game isn't something that can be made by one person alone. Well, it's possible that a really hard-working game creator could finish a game all alone, but even then he would still need a player. Games grow and mature when they're created, played, and conveyed over and over.

Shigesato Itoi

RESUMEN

El objeto de este trabajo de fin de grado es el desarrollo de una aplicación de escritorio que facilite a los desarrolladores de videojuegos la tarea del diseño de niveles para cualquier videojuego que de una forma u otra contengan lo que en la industria se denomina como “mazmorra”.

Esta aplicación tendrá una interfaz gráfica en la que los usuarios podrán construir los espacios jugables, definir qué regiones de los mismos conforman los niveles, visualizar la totalidad de su diseño como una gran estructura conjunta y crear instancias de una serie de objetos lógicos para dejar una estructura lógica implícita en el diseño.

Además, la herramienta creada deberá poder ser integrada en el proceso de desarrollo. Exportando los diseños contruidos por el usuario a un motor de videojuegos donde los desarrolladores podrán continuar con el trabajo en su producto.

Finalmente, esta aplicación utilizará resultados de la teoría de grafos y algunas técnicas del ámbito de la inteligencia artificial para explorar los diseños creados por los usuarios y así comprobar que son correctos. Los criterios de corrección asegurarían que no existe ninguna sucesión de decisiones que un jugador podría tomar cuya consecuencia sea que éste no pueda completar el juego o que algunas áreas del mismo queden permanentemente inaccesibles.

Durante este trabajo se discutirán las herramientas actualmente disponibles, la motivación detrás del desarrollo de ésta herramienta, la definición del propio proyecto, su diseño como producto software, su desarrollo, las decisiones tomadas durante cada una de éstas fases, las conclusiones, los resultados obtenidos y posibles mejoras o ampliaciones futuras que podrían aplicarse al producto.

PALABRAS CLAVE

aplicación de escritorio, videojuegos, diseño, interfaz gráfica, espacios jugables, niveles, instanciar, objetos lógicos, estructura lógica implícita, motor de videojuegos, teoría de grafos, inteligencia artificial

ABSTRACT

The purpose of this document is the development of a desktop application that eases video game developers the task of level design for any game that in one way or another contain what is known as a “maze” in the industry.

This application will have a graphical interface in which users will be able to build the playable areas, define which regions of those areas make up the levels, visualize the totality of their design as one big connected structure and instantiate a set of logic objects that will leave an underlying implicit logical structure in the design.

The tool created will be able to be integrated in the development process. Exporting the designs built by the users into a video game engine in which the developers will be able to continue working on their product.

Finally, this application will use graph theory's results and some techniques from the field of artificial intelligence to explore the designs created by the users in order to ensure that they are correct. the criteria for correction will make sure that there does not exist any succession of decisions that as consequence have the player not being able to finish the game or leave some áreas of it permanently unreachable.

In this document the current tools available, the motivation behind the development of this tool, the definition of the project itself, its design as a software product, its development, the decisions taken during all the stages of development, the conclusions, the results and potential future upgrades or updates will be discussed.

KEYWORDS

desktop application, video game, design, graphical interface, playable areas, levels, instantiate, logic objects, implicit logical structure, video game engine, graph theory, artificial intelligence

ÍNDICE

1	Introducción	1
1.1	Motivación	1
1.2	Objetivos	4
1.3	Estructura	4
2	Estado del Arte	5
2.1	Videojuegos de tipo Mazmorra	6
2.2	Diseño Gráfico	6
2.3	Diseño Lógico	7
2.4	Motores y Entornos	7
2.5	C++ y Qt	8
2.6	Conclusiones	9
3	Definición del proyecto	11
3.1	Alcance	11
3.2	Requisitos funcionales	11
3.2.1	Requisitos de persistencia	12
3.2.2	Requisitos de contexto	12
3.2.3	Requisitos del canvas	12
3.2.4	Requisitos de llaves	13
3.2.5	Requisitos de condiciones de apertura	14
3.2.6	Requisitos de comprobación de la corrección	14
3.2.7	Requisitos de conexión con Game Maker Studio 2	14
3.3	Requisitos no funcionales	15
3.3.1	Requisitos de sistemas	15
4	Diseño	17
4.1	Tecnologías y Estándares	17
4.1.1	Qt	17
4.1.2	JSON	19
4.1.3	git	19
4.1.4	GameMaker Studio 2	19
4.2	Módulos y Clases	20
5	Desarrollo	23

5.1 Entornos de desarrollo	23
5.2 Desarrollo de los módulos	23
5.2.1 Canvas	24
5.2.2 Llaves, habitaciones y persistencia	25
5.2.3 Interfaz gráfica	27
5.2.4 Condiciones y exploración del diseño	29
5.3 Dificultades encontradas	32
6 Integración, pruebas y resultados	35
6.1 Pruebas unitarias	35
6.2 Pruebas de integración	36
6.3 Resultados	36
7 Conclusiones y trabajo futuro	37
7.1 Conclusiones	37
7.2 Trabajo futuro	37
Bibliografía	39
Definiciones	41
Apéndices	43
A Definición de la sintaxis de condiciones	45
B Patrón de diseño de niveles empleado como tutorial	47

LISTAS

Lista de códigos

A.1	definición de la sintaxis	45
-----	---------------------------------	----

Lista de figuras

1.1	2
1.2	3
2.1	5
4.1	18
4.2	21
4.3	22
5.1	24
5.2	24
5.3	25
5.4	25
5.5	26
5.6	26
5.7	26
5.8	27
5.9	28
5.10	28
5.11	28
5.12	29
5.13	31
5.14	32
5.15	33
B.1	47
B.2	48

INTRODUCCIÓN

Este Trabajo de Fin de Grado tiene como propósito el desarrollo de una aplicación de escritorio que servirá de herramienta para facilitar el desarrollo de **videojuegos de tipo mazmorra**.

La aplicación de escritorio *Maze Designer* se encargará de la tarea del diseño de **niveles**, permitiendo visualizarlos y editarlos como una gran estructura conjunta en lugar de como escenarios aislados y facilitando una interfaz lógica que compruebe la corrección de un diseño.

1.1. Motivación

En la actualidad, el género de videojuegos de tipo mazmorra ha adquirido bastante popularidad tanto entre grandes desarrolladoras como en el escenario independiente del desarrollo de videojuegos. Se han creado títulos como *Super Metroid*¹ (Nintendo, 1994), *Cave Story*² (Pixel, Nicalis, 2004), *Ori and the Blind Forest*³ (Moon Studios, 2015) y *Hollow Knight*⁴ (Team Cherry, 2017) (ver figura 1.1) entre muchos otros, que a día de hoy se consideran referentes e incluso juegos de culto.

Este tipo de juegos, centrados en la exploración y avance a través de un entorno, presentan ciertas necesidades particulares a la hora de diseñar los **espacios de juego** al ser éstos un elemento tan importante para la experiencia del jugador. Además cumplir con los requisitos visuales y estéticos del título en cuestión, los espacios de juego se diseñan de manera que el jugador necesite realizar ciertos logros, como avanzar en la trama u obtener un cierto objeto, para progresar y poder explorar nuevas zonas. De esta manera, los espacios de juego son diseñados para controlar el progreso del jugador, guiándole (sin que éste se de cuenta) hacia una experiencia determinada.

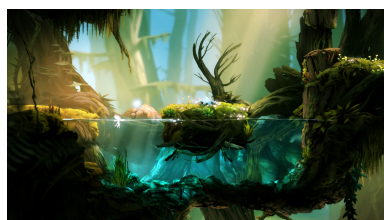
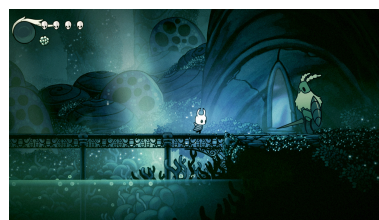
Para esto, se suelen emplear distintos patrones de bloqueo para dejar ciertas áreas inaccesibles hasta cumplir con unos requisitos determinados.

¹ Nintendo's page for Super Metroid: <https://www.nintendo.es/Juegos/Super-Nintendo/Super-Metroid-279613.html>

² Cave Story - Doukutsu Monogatari: <https://www.cavestory.org/>

³ Ori And The Blind Forest main page: <https://www.orithegame.com/blind-forest/>

⁴ Hollow Knight main page: <https://hollowknight.com/>

(a) *Super Metroid* (Nintendo, 1994)(b) *Cave Story* (Pixel, Nicalis, 2004)(c) *Ori and the Blind Forest* (Moon Studios, 2015)(d) *Hollow Knight* (Team Cherry, 2017)**Figura 1.1:** Algunos de los videojuegos de tipo mazmorra notables.

Estos bloqueos no son necesariamente bidireccionales, de manera que se le permite al jugador acceder a un área pero regresar a la zona anterior inmediatamente no es necesariamente posible. Éste es uno de los patrones más populares, empleado usualmente a modo de tutorial (como se ilustra en el apéndice B), pero utilizado de forma errónea podría ocasionar que el jugador quedara atrapado en un área sin poder salir de la misma.

Además de esto, los patrones para el desbloqueo de áreas más extendidos son el de *power-ups* y el de llaves comportándose como *objetos consumibles*. En particular este último también puede plantear situaciones en las que algunas áreas queden permanentemente inaccesibles para el jugador si los espacios de juego están diseñados erróneamente y éste los explora de una determinada manera.

Debido los patrones problemáticos antes mencionados, el diseño de espacios de juego se vuelve muy costoso conforme éstos aumentan de tamaño, y más aún si el progreso del juego no es lineal. esto se puede observar en la figura 1.2. Por esto se plantea el desarrollo de una herramienta que facilite al diseñador la tarea de crear estos espacios de juego y, una vez acabados, los explore para asegurar el correcto diseño de los mismos. Es decir, que el jugador no pueda realizar una exploración que como consecuencia le deje permanentemente atascado o que deje un área inaccesible sin remedio.

La intención de este trabajo de fin de grado es desarrollar tal herramienta para facilitar el desarrollo de videojuegos de tipo mazmorra.

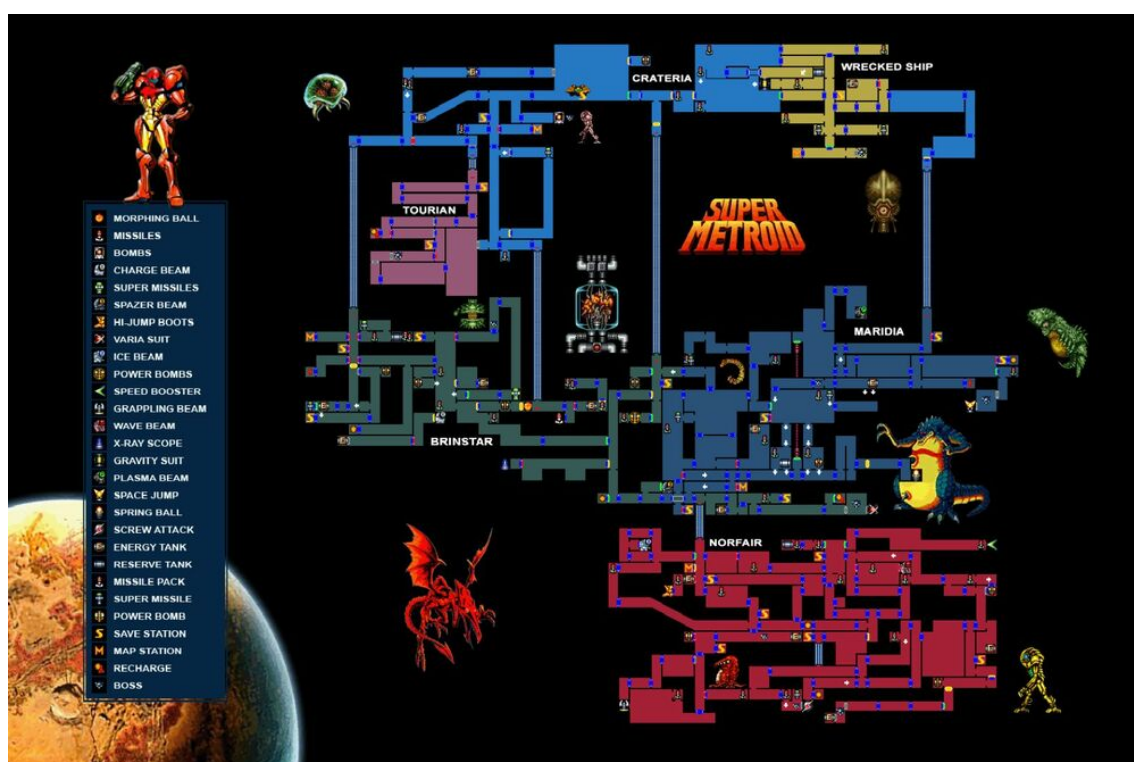


Figura 1.2: mapa del videojuego *Super Metroid* (Nintendo, 1994) en el que se reflejan todos los espacios jugables, todos los objetos coleccionables tal y como se indica en la leyenda del mapa. Además, las puertas en el mapa están representadas por un código de colores con el que se identifica el *power-up* requerido para abrirlas (funcionando así como elementos bloqueadores).

1.2. Objetivos

El proyecto se centra en el desarrollo de una aplicación de escritorio que sirva como entorno de diseño de niveles de cualquier videojuego que, de una forma u otra, contenga una **mazmorra**.

La herramienta debe ser capaz de explorar el diseño creado por el usuario para asegurar que éste no presenta errores en el progreso del jugador y debe exportar los diseños a un motor de videojuegos escogido (*GameMaker Studio 2*) para continuar con el desarrollo del título en él.

Los objetivos principales a cumplir por la herramienta son:

- 1.– Una interfaz para editar el espacio jugable de forma visual.
- 2.– Una interfaz para delimitar los niveles del juego que más tarde serán exportados al motor de videojuegos.
- 3.– Una interfaz para colocar elementos del progreso lógico del jugador (llaves, bloqueos y punto de comienzo).
- 4.– La exportación del diseño elaborado por el usuario al motor *GameMaker Studio 2* para continuar en éste el desarrollo del videojuego.
- 5.– La exploración del diseño para la detección de errores lógicos en el progreso del jugador diseñado implícitamente.

Para ello, se han desarrollado los principales módulos de la aplicación como proyectos de Qt [1] [2] independientes unos de otros en la medida de lo posible, que finalmente son integrados en una interfaz gráfica.

1.3. Estructura

Esta memoria del Trabajo de Fin de Grado está dividida en 6 capítulos además de esta introducción. Éstos son:

- 1.– **Estado del arte** donde se detalla tecnología existente y herramientas utilizadas para tareas con cierta similitud a las partes de la aplicación a desarrollar, así como tecnologías utilizadas en el desarrollo de videojuegos que tienen directa relación con el producto que se construirá con esta aplicación. También se discutirá en este apartado ventajas e inconvenientes de cada una de las tecnologías que se mencionen.
- 2.– **Análisis de Requisitos** donde se exponen los requisitos mínimos, funcionales y no funcionales, que se han de cumplir por la aplicación, definiendo así el alcance de la misma.
- 3.– **Diseño** donde se plantea el diseño de la aplicación, el diagrama de clases y se discutirán las distintas decisiones de diseño tomadas en el desarrollo.
- 4.– **Desarrollo** donde se detalla el proceso seguido para el desarrollo, hablando de las metodologías empleadas, de dificultades encontradas y de las soluciones llevadas a cabo.
- 5.– **Integración, pruebas y resultados** donde se discute el resultado de la aplicación y se muestran las pruebas a las que ésta ha sido sometida para asegurar el correcto funcionamiento y el control de la calidad de la misma.
- 6.– **Conclusiones y trabajo futuro** donde se habla del resultado obtenido, la utilidad de la herramienta desarrollada y de posibles mejoras y ampliaciones que podrían llevarse a cabo en el futuro.

ESTADO DEL ARTE

Históricamente, el rol del diseñador de este tipo de juegos comenzaba dibujando en papel el diseño de los mismos (tal y como observamos en la figura 2.1). Se empezaba por un esquema en forma de grafo en el que se describía el progreso lógico del jugador (qué requisitos debe cumplir para acceder al área siguiente) y más tarde se implementaba esta lógica del progreso en los niveles diseñados ya a nivel visual. Más adelante, se empezaron a utilizar herramientas software para sustituir al papel, pero las tareas permanecían divididas en un diseño lógico y visual, más cercano a lo que aparece en el producto final. Además de esto, la traducción de estos diseños a componentes del juego se ha llevado a cabo manualmente en distintos entornos de desarrollo y, en la actualidad, se utilizan los motores de videojuegos para este propósito.

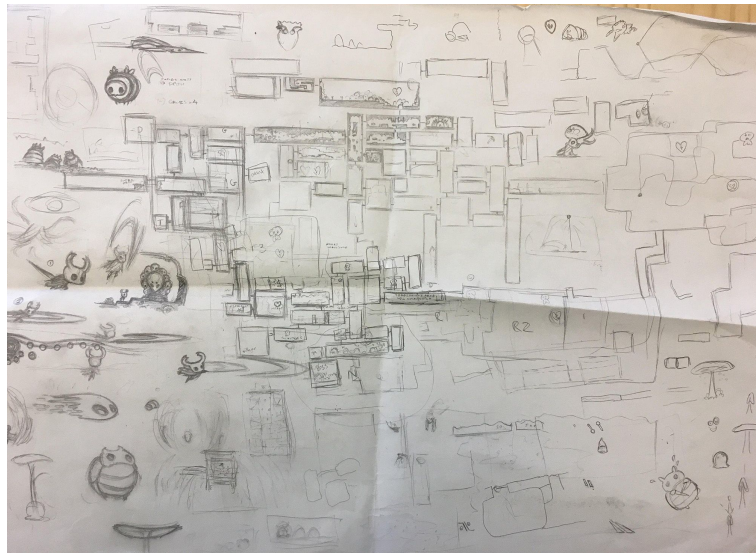


Figura 2.1: Sketch realizado por Ari Gibson. Es un boceto del mapa del videojuego *Hollow Knight* (Team Cherry, 2017) que se hizo en las fases tempranas del desarrollo.

2.1. Videojuegos de tipo Mazmorra

Los videojuegos de tipo mazmorra, englobando videojuegos que la industria ha bautizado como *metroidvania* y *zelda-like* entre otros, es un género de productos de ocio digital en los que el usuario controla a un avatar en un entorno que debe explorar, usualmente controlando la progresión del jugador a través de dicho entorno mediante puertas bloqueadas u obstáculos que debe sortear consiguiendo llaves o habilidades nuevas para su avatar.

El foco central de estos juegos es la exploración y la progresión a través del escenario, a diferencia de otros géneros de videojuegos, donde el foco puede estar centrado en la trama, en asumir el rol de un determinado personaje o en simular una experiencia como puede ser en un juego de deporte. El diseño de cómo es la exploración depende del desarrollador y de la visión que tenga sobre su producto.

Al hablar de la exploración de estos juegos, se suelen emplear términos como **exploración lineal**, **exploración ramificada** y **backtracking** para describirlos. Durante la exploración de los escenarios del juego, el jugador puede recolectar diversos objetos que, dependiendo del juego, tienen un uso u otro, sin embargo se repiten mucho los patrones de **objeto consumible** y **power-up** ya que estos en particular pueden utilizarse para el control del progreso del jugador.

Algunos de los títulos de éste género más notables de la industria se presentaron en la figura 1.1.

2.2. Diseño Gráfico

Las herramientas de diseño gráfico, que no son capaces de realizar ningún tipo de abstracción lógica, sino que son empleadas únicamente para representar el aspecto del diseño, con el nivel de detalle que el usuario decida. La actual tecnología empleada para este tipo de diseño, varía desde el diseño en papel, hasta herramientas software como:

- 1.– **Photoshop**¹ La herramienta de diseño gráfico profesional de *Adobe*, permite realizar detalladas manipulaciones de imágenes para diseñar el espacio de juego e incluso permite el uso de capas para mayor utilidad.
- 2.– **Gimp**² La herramienta open source de diseño gráfico que cumple, a los efectos del diseño de niveles de un videojuego, los mismos propósitos que *photoshop*
- 3.– **Paint.net**³ La herramienta gratuita de edición de imágenes que, a pesar de ser sencilla, es bastante potente en cuanto a las funcionalidades que ofrece, alcanzando estándares similares a *GIMP*.
- 4.– **Tiled**⁴ Es un editor de mapas orientado específicamente al diseño de videojuegos, pero su función es únicamente la colocación de elementos gráficos para la construcción de los niveles.

Todas las herramientas mencionadas presentan el problema en común de que no aplican ningún

¹ Photoshop main page: <https://www.photoshop.com/>

² GIMP - The Free and Open Source Image Editor: <https://www.gimp.org/>

³ Paint dot net main page: <https://www.getpaint.net/>

⁴ Tiled - your free, easy to use and flexible level editor: <https://www.mapeditor.org/>

tipo de abstracción lógica al diseño, de manera que la comprobación de que un éste sea correcto queda a discreción del usuario. Además de esto, la integración de los diseños con un motor de videojuegos en específico, a excepción de *Tiled* es íntegramente manual.

2.3. Diseño Lógico

Las herramientas del diseño lógico se basan en la abstracción del progreso del jugador en forma de grafo. Dado que el progreso se modela como un **grafo dirigido**, la tecnología orientada a cubrir estos requerimientos consiste en software de visualización y manipulación de **grafo**. Para estas tareas específicas, existen las siguientes herramientas:

- 1.– **graphviz**⁵ Se trata de una aplicación para diseñar y visualizar información abstracta estructurada en grafos.
- 2.– **librería diagrams de Haskell**⁶ Se trata de una librería de Haskell con la que se pueden declarar y visualizar grafos. Es una potente librería de fácil utilización, sin embargo tiene el defecto de ser necesario programar en el lenguaje de programación Haskell, lo cual es una tarea que puede escaparse de las competencias de un diseñador o puede llegar a ser demasiado complejo para la tarea en cuestión.
- 3.– **Finite State Machine Designer**⁷ Se trata de una aplicación web para diseñar grafos de forma visual cuyo objetivo principal es el diseño de autómatas finitos. Sin embargo, debido a la naturaleza de grafo de los diseños realizados y a la sencillez de utilización, puede cumplir con los requisitos de esta categoría de diseño, aunque tratándose de una aplicación web, puede resultar incómoda en especial para diseños muy complejos.

Todas las herramientas mencionadas comparten el defecto de ser puramente abstractas para representar el progreso en forma de grafo, sin preocuparse de la visualización de los niveles y, por tanto, la implementación de la lógica diseñada con estas herramientas a niveles de un producto es una tarea que el diseñador debe realizar manualmente.

2.4. Motores y Entornos

En la actualidad la producción de videojuegos se lleva a cabo en motores de videojuegos, que conforman un entorno de desarrollo. Algunos de estos motores son de propósito general y permiten el desarrollo de cualquier tipo de juego.

- 1.– **Construct**⁸ Es un motor de videojuegos creado por *Scirra Ltd*. Se trata de un motor de propósito general orientado a un público con pocos conocimientos técnicos en el desarrollo de videojuegos. Permite crear videojuegos en dos dimensiones.
- 2.– **framework XNA4 [3]** Es un framework desarrollado por *Microsoft* para el desarrollo de videojuegos. Fue descontinuado en 2010
- 3.– **MonoGame**⁹ Es una implementación del framework XNA4 de Microsoft, desarrollada por *MonoGame*

⁵Graphviz - Graph Visualization Software: <http://www.graphviz.org/>

⁶diagrams: Embedded domain-specific language for declarative vector graphics: <http://hackage.haskell.org/package/diagrams>

⁷Finite State Machine Designer: <http://madebyevan.com/fsm/>

⁸Construct 3 - Game Making Software: <https://www.construct.net/en>

⁹MonoGame - One framework for creating powerful cross-platform games: <http://www.monogame.net/>

Team. Se trata de una herramienta open-source que permite la creación de videojuegos tanto en dos como en tres dimensiones.

4.– **Unity**¹⁰ Es un motor de videojuegos creado por *Unity technologies*. Se trata de un motor de propósito general que permite la creación de videojuegos tanto en tres como en dos dimensiones.

5.– **Unreal Engine**¹¹ Es un motor de videojuegos creado por *Epic Games*. Es también un motor de propósito general para creación de juegos tanto en tres como en dos dimensiones.

6.– **GameMaker Studio 2** [4] Es un motor de videojuegos creado por *YoYo Games*. También de propósito general pero limitado a la creación de juegos en 2D.

De entre los motores enumerados, se ha escogido *GameMaker Studio 2* por ser de propósito general, suficientemente completo. Al limitar su ámbito a los juegos en dos dimensiones, se simplifican muchas de las complicaciones que presentan los juegos en tres dimensiones, entre las que se encuentra la creación de escenarios, que es el propósito principal que abarca la herramienta *Maze Designer*.

2.5. C++ y Qt

C++ [5] [6] es un lenguaje de programación multiparadigma cuyo origen es la extensión del lenguaje de programación C al paradigma de la **orientación a objetos**. Tras esto, conforme C++ ha ido evolucionando, se han incorporado facultades propias de otros paradigmas, de manera que ahora engloba a la **programación estructurada**, a la **orientación a objetos**, a la **programación genérica** y en las versiones más actualizadas contiene ciertos conceptos de **programación funcional**. Como características distintivas, C++ permite al programador el uso de punteros, gestión de la memoria y también permite la sobrecarga de operadores. Actualmente, C++ se encuentra en su versión C++17, destacando la versión C++11 que realizó mejoras a nivel del núcleo del lenguaje. C++ es un lenguaje compilado que destaca por su alto rendimiento, sus capacidades para paralelizar tareas y por la gestión de la memoria por parte del programador.

Junto a C++, se encuentra Qt [7] [1] [2], un framework que no pertenece al estándar pero de uso muy extendido para la tarea del desarrollo de aplicaciones de escritorio. Qt incluye librerías y APIs escritas en C++ que abarca ámbitos desde renderización, cálculos geométricos, multithreading, control del input de dispositivos hardware y hasta realización de peticiones http. Actualmente, Qt está en su versión 5.12 y ofrece un lenguaje propio para declaración de componentes gráficos (QML [8]) y un binding con el lenguaje de programación python.

¹⁰Unity for all: <https://unity.com>

¹¹Unreal Engine - Make something unreal with the most powerful creation engine: <https://www.unrealengine.com/en-US/>

2.6. Conclusiones

Las herramientas orientadas al diseño lógico no cubren los requisitos estéticos y visuales que sí cumplen aquellas orientadas al diseño gráfico, sin embargo estas últimas carecen de la abstracción lógica que aportan las primeras. En cuanto a los motores, al ser de propósito general y no existir ninguno para un género tan concreto como los juegos de tipo mazmorra, no cubren las necesidades de diseño de éstos juegos tan específicamente, dejando la tarea del diseño a los desarrolladores sin proporcionarles ningún tipo de ayuda.

Es por esto que la herramienta *Maze Designer* se plantea como una opción que integra la abstracción lógica de las herramientas de manipulación de grafos (de forma transparente para el diseñador) con el cumplimiento de ciertos requisitos visuales que cumplen las herramientas de diseño gráfico, ofreciendo además una conexión a un motor de videojuegos donde se puede continuar con el desarrollo una vez terminado el diseño de los espacios de juego.

DEFINICIÓN DEL PROYECTO

3.1. Alcance

La aplicación será una herramienta de diseño de niveles, por tanto abarcará las tareas de construcción de espacios, la instanciación (solamente en el diseño) de agentes lógicos como llaves y puertas; y la posterior exploración del diseño para la comprobación de que éste es correcto, es decir, no existe una cadena de acciones o exploración que tiene como consecuencia que el jugador no pueda acceder a algún área del juego.

De la misma manera, la aplicación deberá poder exportar estos diseños a un proyecto de *Game-Maker Studio 2*. Esta exportación creará tantos ficheros .yy (utilizados por el motor para representar los niveles) como habitaciones hayan sido definidas en el diseño. Al contenido de dichos ficheros .yy solo se exportarán los objetos sólidos que conforman las paredes, suelos y techos de la habitación en cuestión. Es decir, las instancias lógicas antes mencionadas no serán exportadas al fichero .yy dado que en la implementación del juego serán objetos cuyo funcionamiento deberá implementar el desarrollador.

Esta herramienta no es un editor de niveles, pues no cubre la funcionalidad de colocar instancias en los espacios de juego.

Además de esto, a la hora de realizar la exploración, solamente se tienen en cuenta las componente conexas del diseño funcionando como nodos y las puertas funcionando como aristas. Es decir, la herramienta no tendrá en cuenta mecánicas de juego como “el jugador no puede saltar más de una altura X y por tanto no puede llegar a determinados lugares” a la hora de determinar qué áreas son accesibles o no. Modelar algo así en esta herramienta es tarea del desarrollador.

3.2. Requisitos funcionales

3.2.1. Requisitos de persistencia

RF-1.– Guardar el diseño.

El usuario debe poder guardar los datos del diseño sobre el que está trabajando, especificando el nombre del fichero. Dicho fichero se guardará con la extensión *.maze*.

RF-2.– Cargar el diseño.

El usuario debe poder cargar los datos de un diseño previamente guardado.

RF-3.– Crear nuevo diseño.

El usuario debe poder crear un nuevo diseño vacío sobre el que trabajar.

3.2.2. Requisitos de contexto

RF-4.– Cambio de contexto del canvas.

El usuario podrá cambiar el contexto actual del canvas para realizar tareas diferentes, de manera que la interfaz se adapte a dicho contexto:

RF-4.1.– Contexto de edición de espacios jugables.

En este contexto el usuario podrá realizar todas las operaciones que tengan que ver con la creación, edición, selección y eliminación de los *espacios jugables*.

RF-4.2.– Contexto de edición de pantallas.

En este contexto el usuario podrá realizar todas las operaciones que tengan que ver con la creación, selección y eliminación de las *pantallas*.

RF-4.3.– Contexto de edición de elementos llave y de elementos bloqueadores.

En este contexto el usuario podrá realizar todas las operaciones que tengan que ver con la creación, selección y eliminación tanto de elementos llave como de elementos bloqueadores.

3.2.3. Requisitos del canvas

RF-5.– Dibujar espacios jugables.

El usuario, en el contexto de diseño, debe poder dibujar *espacios jugables* en el *canvas* haciendo *click* en un punto del *canvas* y soltando el *click* en otro.

RF-5.1.– Anexión de espacios jugables.

Si el conjunto de los *espacios jugables* que intersecan con el recién creado es no vacío, entonces se realizará la unión de los *espacios jugables* preexistentes y el nuevo *espacio jugable*

RF-5.2.– Indexación de espacios jugables.

Todo *espacio jugable* debe ser indexado por una estructura para su futura transformación en grafo.

RF-5.3.– Grid de coordenadas.

Los vértices de todo polígono que conforme un *espacio jugable* deben estar ajustados a una rejilla de coordenadas.

RF-6.– Eliminación de espacios jugables.

El usuario debe poder extraer áreas rectangulares de un *espacio jugable*, de manera que el resultado sea un polígono con ángulos de 90 grados o de 270 grados.

RF-7.– Crear un elemento bloqueador.

El usuario debe poder crear un elemento bloqueador dibujando una línea en el *canvas*.

RF-7.1.– Condición de apertura. Todo elemento bloqueador debe tener asignada una condición de apertura. Por defecto, la condición vacía.

RF-7.2.– Localización válida para elementos bloqueadores.

El elemento bloqueador creado debe estar contenido en su totalidad dentro de los espacios jugables.

RF-7.3.– Direcciones permitidas para elementos bloqueadores.

El elemento bloqueador creado solo puede tener una dirección totalmente vertical o totalmente horizontal.

RF-7.4.– Cálculo de vecindad al crear un elemento bloqueador.

A la hora de realizar una exploración, en el caso de que la creación de un elemento bloqueador divida un **espacio jugable** en dos, se han de indexar estos dos nuevos **espacios jugables** y se ha de definir su vecindad teniendo en cuenta el tipo de elemento bloqueador y sus condiciones de apertura.

RF-8.– Seleccionar un elemento bloqueador.

El usuario debe poder seleccionar un elemento bloqueador preexistente del *canvas*.

RF-9.– Eliminar un elemento bloqueador.

El usuario debe poder eliminar un elemento bloqueador preexistente del *canvas*.

RF-10.– Crear un elemento llave.

El usuario debe poder crear un elemento llave, instanciando un modelo de llave en el *canvas*.

RF-11.– Seleccionar un elemento llave.

El usuario debe poder seleccionar una instancia de llave del *canvas*.

RF-12.– Eliminar un elemento llave.

El usuario debe poder eliminar una instancia de llave del *canvas*.

RF-13.– Zoom.

El usuario debe poder acercar y alejar la vista que tiene del *canvas*.

RF-14.– Desplazar vista.

El usuario debe poder desplazar la vista que tiene del *canvas*.

3.2.4. Requisitos de llaves

RF-15.– Crear tipos de llaves.

El usuario debe poder crear modelos de llaves especificando su nombre y si es un **power-up** o no.

RF-16.– Eliminar tipos de llaves.

El usuario debe poder eliminar modelos de llaves previamente creados.

RF-17.– Modificar el nombre de un tipo de llave.

El usuario debe poder modificar el nombre que se le hubiera asignado a un modelo de llave.

RF-18.– Modificar la clase de un tipo de llave.

El usuario debe poder modificar la clase de un modelo de llave, es decir, modificar si es o no un **power-up**.

RF-19.– Seleccionar un tipo de llave.

El usuario debe poder seleccionar un modelo de llave de entre la lista de todos los tipos de llaves.

RF-20.– Unicidad en los nombres de las llaves.

Dos modelos de llaves no podrán compartir nombre. Si el usuario intenta crear una llave con el mismo nombre que otra, se cambiará el nombre, concatenando un número al final del mismo, para que no coincidan.

RF-21.– Caracteres permitidos en los nombres de las llaves.

Los nombres de las llaves solo podrán estar compuestos de por caracteres alfanuméricos, junto con los caracteres “-” y “_”. Si el usuario intenta crear una llave conteniendo otros caracteres, no se efectuará el cambio de nombre.

RF-22.– Nombres de llaves no permitidos.

Las llaves no podrán ser nombradas solamente con números ni usando la palabra clave “have” por pertenecer al lenguaje declarativo de las condiciones.

3.2.5. Requisitos de condiciones de apertura

RF-23.– Edición de condiciones de apertura.

Cuando el usuario haya seleccionado un elemento bloqueador, debe ser capaz de describir las condiciones bajo las cuales este se abrirá.

RF-23.1.– Sintaxis del lenguaje.

Las condiciones deben seguir una sintaxis determinada (ver apéndice A)

RF-23.2.– Condición vacía.

La condición vacía del lenguaje se interpreta como una satisfacible siempre.

RF-23.3.– Condiciones no satisfacibles

Deben poder crearse condiciones no satisfacibles.

RF-23.4.– Orden de precedencia.

A la hora de evaluar una expresión del lenguaje, se debe respetar el orden de precedencia habitual: resolución del contenido de los paréntesis primero y conexión de izquierda a derecha.

RF-24.– Sintetización de condiciones.

El sistema debe ser capaz de sintetizar las condiciones escritas por el usuario para ser asignadas a otros elementos del sistema.

RF-25.– Listas de costes

Una condición debe poder calcular la lista o listas de objetos llave requeridos para ser satisfecha.

3.2.6. Requisitos de comprobación de la corrección

RF-26.– Comprobación.

El usuario debe poder comprobar que su diseño no contenga errores de tipo *dead lock*.

RF-26.1.– Conversión a grafo.

El diseño se debe poder convertir a una estructura de tipo grafo para poder realizar la exploración del mismo

RF-26.2.– Notificación de errores.

Si el diseño comprobado contiene un error de tipo *dead lock* se ha de notificar al usuario del camino escogido para encontrar dicho error.

RF-26.3.– condición de correctitud.

Un diseño es correcto si, realizando cualquier exploración, no es posible llegar a un estado de tipo *dead lock*.

3.2.7. Requisitos de conexión con Game Maker Studio 2

RF-27.– Creación de pantallas.

El usuario debe poder seleccionar regiones rectangulares del canvas que representarán las pantallas del juego que está diseñando. Al crearse, las pantallas tendrán un nombre asignado por defecto.

RF-28.– Seleccionar una pantalla.

El usuario debe poder seleccionar una pantalla.

RF-29.– Cambiar el nombre de una pantalla.

El usuario debe poder cambiar el nombre de una pantalla.

RF-30.– Eliminar una pantalla.

El usuario debe poder eliminar un región que representa una pantalla de su juego

RF-31.– Exportar pantallas.

El usuario debe poder elegir qué pantallas son exportadas y cuáles no lo son.

RF-32.– Conversión a ficheros .yy.

El sistema debe ser capaz de transformar las pantallas a ficheros .yy compatibles con un proyecto de *Game Maker Studio 2*, con las dimensiones y nombre determinados por el diseño del usuario. En estos ficheros figurarán solamente los bloques sólidos que cubrirán los espacios no jugables del diseño que haya dentro de cada una de las pantallas.

RF-32.1.– Construcción de un área con el mínimo número de rectángulos.

Los espacios no jugables, que tendrán forma poligonal cuyos ángulos serán múltiplos de 90 grados, han de cubrirse del menor número de rectángulos posible. Estos rectángulos, en la traducción a ficheros .yy, se representarán como bloques sólidos del juego.

3.3. Requisitos no funcionales

3.3.1. Requisitos de sistemas

RNF-1.– Sistemas operativos compatibles.

La aplicación debe poder ejecutarse al menos en *Windows*.

RNF-2.– Compatibilidad con gestores de versiones.

Los archivos generados por la aplicación, almacenando los datos de persistencia de un diseño, deben ser compatibles con un gestor de versiones como git [9]

RNF-3.– Legibilidad de los archivos.

Los archivos generados por la aplicación, almacenando los datos de persistencia de un diseño, deben poder ser leídos por un desarrollador.

DISEÑO

4.1. Tecnologías y Estándares

Para el desarrollo de esta aplicación, ha sido necesario el uso de varios estándares y tecnologías que interactuasen entre sí. La codificación se desarrolla en *C++* junto con *Qt*, pero se debía atender a necesidades de compatibilidad con *GameMaker Studio 2*, mediante el estándar *JSON*, así como respetar un formato de persistencia que fuera amigable con un sistema de control de versiones como *git*. Así pues, el conjunto de las tecnologías empleadas y cómo han sido relacionadas se explica en la figura 4.1.

4.1.1. Qt

Qt [1] [2] [7] es el framework de *C++* empleado para la implementación, tanto lógica como de interfaz de usuario, de esta aplicación.

Se han empleado diversas clases de *Qt* para la codificación de la aplicación, desde clases generales como *QString*, *QList*, etc. hasta clases más específicas de ciertos ámbitos como *QRectF*, *QPolygonF*, *QPainterPath* para las operaciones geométricas, *QJsonObject*, *QJsonDocument* para las operaciones de persistencia y generación de ficheros *.yy* y distintas clases que heredan de *QWidget* y *QLayout* para la construcción de la interfaz gráfica de usuario. También se ha hecho uso del sistema de slots y señales que proporciona *Qt* para la comunicación entre las distintas componentes de la aplicación y para la interacción con el usuario. Este sistema sustituye a los callbacks usuales en el desarrollo de aplicaciones de escritorio.

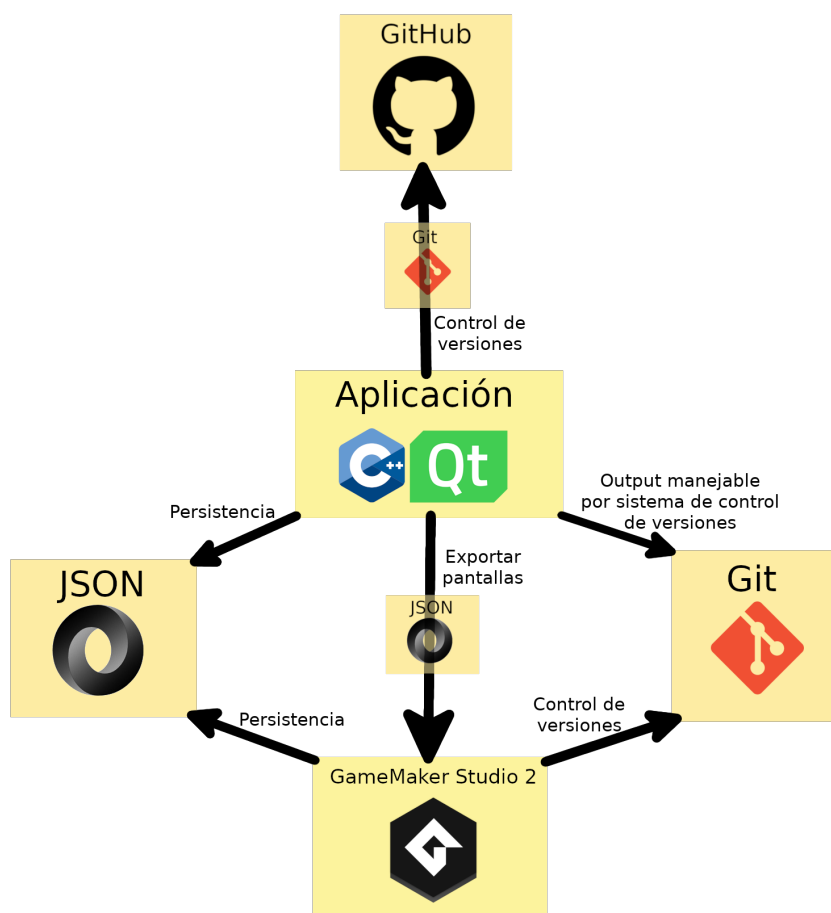


Figura 4.1: Diagrama representando las tecnologías que intervienen en el proyecto y las interacciones entre ellas.

4.1.2. JSON

JSON [10] es un estándar ¹ de formato de descripción de objetos para el intercambio de datos. se ha empleado este estándar en los módulos de persistencia, para guardar los datos en ficheros de manera que luego puedan ser leídos por el software para reconstruir los datos de una sesión anterior. *Game Maker Studio 2* también utiliza este formato para mantener, en ficheros separados, la persistencia de los proyectos desarrollados con dicho motor, por lo que el módulo de esta aplicación que se encarga de exportar datos del diseño, debe hacer uso del estándar JSON.

Adicionalmente, debido a cómo se generan los datos, el formato JSON debidamente escrito en ficheros es amigable tanto para los desarrolladores, que pueden leer ficheros escritos en dicho formato, como con tecnologías como git para el control de versiones.

4.1.3. git

Git [9] es una herramienta open source de control de versiones diseñado para manejar todo tipo de proyectos software de forma rápida y eficiente. Junto con git, existen múltiples plataformas de repositorios que pueden emplearse para guardar los avances del desarrollo de forma remota. Tales como *github* ² (la plataforma escogida para esta aplicación) o *bitbucket* ³. Durante el desarrollo de esta aplicación, se ha utilizado para el control de versiones de la misma, pero también se ha tenido en cuenta a la hora de construir un sistema de persistencia que fuera amigable con git, dado que la aplicación desarrollada es una herramienta para desarrollar otros productos, es deseable que el desarrollo de estos productos últimos pudiera también manejarse con una herramienta de control de versiones.

4.1.4. GameMaker Studio 2

Game Maker Studio 2 [4] es un motor de videojuegos desarrollado por YoYo Games de propósito general y en el cual se pueden desarrollar videojuegos en dos dimensiones. En el desarrollo de esta aplicación, se ha tenido en cuenta como el eslabón siguiente a *Maze Designer* en el *pipeline* del desarrollo de un videojuego. Es decir, que los productos creados utilizando la herramienta que se desarrolla en este trabajo de fin de grado están preparados para formar parte de un proyecto de *Game Maker Studio 2* Los proyectos de *Game Maker Studio 2* son guardados en disco como un conjunto de directorios y ficheros, cuyo contenido sigue el formato *JSON* anteriormente descrito.

¹The JSON Data Interchange Syntax: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

²The world's leading software development platform · GitHub: <https://github.com/>

³Bitbucket — The Git solution for professional teams: <https://bitbucket.org/>

4.2. Módulos y Clases

Los módulos que componen la aplicación atienden a las distintas áreas de los requisitos funcionales, cada uno cubriendo varios de ellos. Estos módulos son:

- **Llaves** que se encarga de cubrir todas las funcionalidades descritas en la subsección 3.2.4. Este módulo permite la creación, edición y eliminación de modelos de llaves que serán utilizados más adelante por otros módulos.
- **Habitaciones** que se encarga de cubrir las funcionalidades descritas en la subsección 3.2.7 a excepción del requisito 32. Este módulo se permite la declaración de regiones del espacio jugable como niveles, que serán más adelante exportados a ficheros de *GameMaker Studio 2*. También permite renombrar dichas habitaciones, eliminarlas y definir si serán o no exportadas.
- **Canvas** que se encarga de cubrir todas las funcionalidades descritas en la subsección 3.2.3. Para esto se han desarrollado 4 submódulos:
 - **Grid** que se encarga de la gestión espacial del módulo, esto es, traduce coordenadas del cursor obtenidas a través del sistema de eventos de Qt a coordenadas lógicas del diseño, aplicando transformaciones de desplazamiento y de zoom, a la vez que provee de una rejilla que se utiliza como base en el resto de los submódulos del *canvas*.
 - **Design Canvas** que se encarga de todas las funcionalidades referentes a la creación de *espacios jugables*. Este submódulo depende del submódulo Grid.
 - **Room Canvas** que permite la declaración de ciertas regiones del diseño como *pantallas*. Este submódulo depende del módulo Habitaciones así como del submódulo Grid.
 - **Instance Canvas** que permite la instanciación de modelos de llaves en el canvas, así como la de puertas y de un token de comienzo a que sirve como punto de partida para la exploración. Este submódulo depende del módulo Llaves así como del submódulo Grid.
- **Condiciones** que se encarga de cubrir las funcionalidades descritas en la subsección 3.2.5. Este módulo permite la interpretación de condiciones escritas por el usuario siguiendo una sintaxis particular (ver apéndice A).
- **Exportación** que se encarga de cubrir la funcionalidad del requisito 32, es decir, permite exportar los diseños creados en la aplicación a ficheros que el motor *GameMaker Studio 2* utiliza como *pantallas*.
- **Exploración** que se encarga de cubrir las funcionalidades descritas en la subsección 3.2.6. Este módulo depende de los módulos Canvas, Condiciones y Llaves.
- **Interfaz Gráfica** que se encarga de integrar todos los módulos en una interfaz de usuario, manifestando las funcionalidades del resto de módulos. Así mismo, la interfaz cumple con las funcionalidades descritas en la subsección 3.2.2.

Las funcionalidades de la subsección 3.2.1 son cubiertas a través de los módulos Canvas, Llaves, Habitaciones y Condiciones.

El esquema representando todos estos módulos junto con sus dependencias se puede observar en la figura 4.2. El diagrama de clases se puede observar en la figura 4.3.

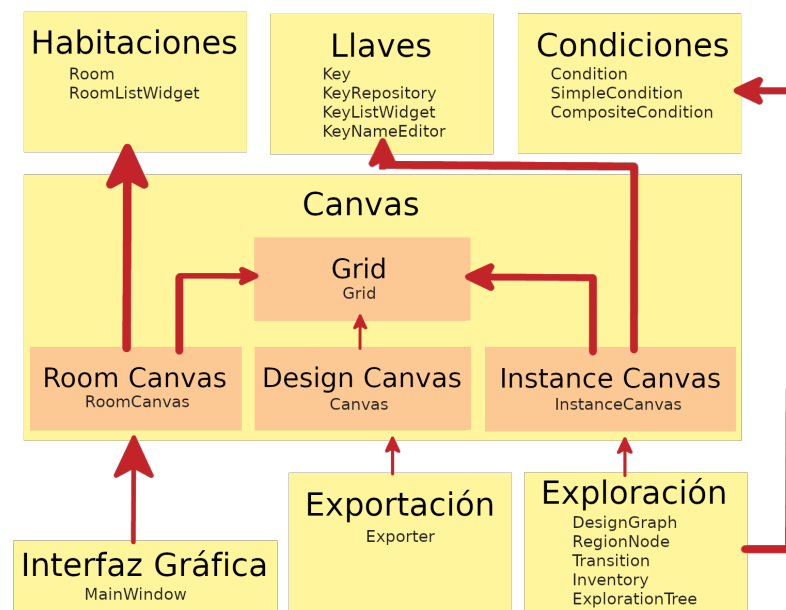


Figura 4.2: Diagrama representando los módulos desarrollados, las clases que éstos contienen y las dependencias entre ellos.



DESARROLLO

Para el desarrollo de esta aplicación se ha seguido un modelo incremental iterativo, dividido en 4 etapas con una etapa previa del análisis de requisitos. Cada una de ellas subdividida en una fase de diseño, una fase de codificación, una fase de pruebas y finalmente una fase de integración.

Las cuatro etapas han sido divididas atendiendo a funcionalidad y son:

- 1.– **Canvas.** En esta etapa se cubren los requisitos relativos a la componente de la aplicación que el usuario utilizaría para la construcción de los **espacios jugables**, así como todos los submódulos requeridos.
- 2.– **Llaves, habitaciones y persistencia.** En esta etapa se cubren los requisitos relativos a la definición de modelos de llaves, la selección de regiones en el canvas para la declaración de habitaciones, el submódulo del canvas para la instanciación de llaves y puertas y la persistencia de todo lo desarrollado hasta el momento.
- 3.– **Interfaz gráfica.** En esta etapa se han integrado todas las componentes hasta ahora desarrolladas y se ha extendido su funcionalidad para la comunicación entre las componentes y la representación de la información de los elementos del diseño.
- 4.– **Condiciones y exploración del diseño.** En esta etapa se cubren los requisitos relativos a la definición de las clases que gestionan las condiciones de apertura de las puertas así como la exploración del diseño para la comprobación de su correctitud.

5.1. Entornos de desarrollo

Para la creación de la aplicación se han empleado distintos entornos de desarrollo. Para la implementación de la aplicación en sí, se ha utilizado *qtCreator*, que es un IDE preparado específicamente para el desarrollo de aplicaciones utilizando C++ y Qt. Se ha investigado el motor de videojuegos *Game Maker Studio 2* [4] tanto a nivel de usuario como su funcionamiento interno en lo que a persistencia y gestión de los recursos de un proyecto se refiere. Para la elaboración de esta memoria, se ha utilizado *TeXstudio* ¹, que se trata de un entorno que facilita la escritura de documentos LaTeX.

5.2. Desarrollo de los módulos

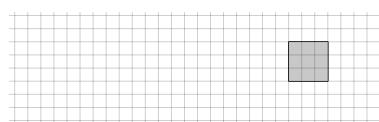
¹integrated writing environment for creating LaTeX documents: <https://www.texstudio.org/>

5.2.1. Canvas

Para el desarrollo del *canvas* y de todos los *sub-canvas* específicos para cada tarea, es necesaria una componente *grid* que represente el espacio del diseño (un sistema de coordenadas) que permita desplazar la vista por el espacio de forma virtualmente ilimitada haciendo click con la rueda del ratón y desplazándolo (tal y como se muestra en la figura 5.1) así como acercar y alejar la vista haciendo girar la rueda del ratón (como se ilustra en la figura 5.2).

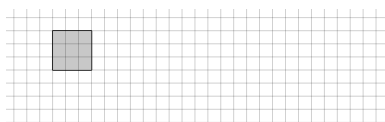


(a) el espacio sin haber realizado una traslación.

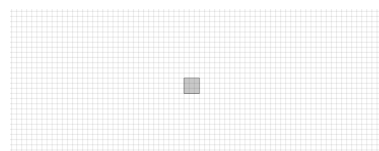


(b) el espacio tras haber realizado una traslación.

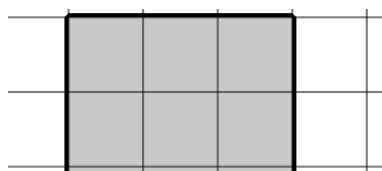
Figura 5.1: Se puede mover, de forma virtualmente ilimitada, la vista que se presenta del espacio de diseño. En estas figuras se ha añadido un espacio jugable para que sirva como referencia.



(a) el espacio sin haber acercado ni alejado la vista.



(b) el espacio tras haber alejado la vista.



(c) el espacio tras haber acercado la vista.

Figura 5.2: Se puede acercar y alejar la vista que se tiene del espacio. El límite está en dos ampliaciones y dos reducciones.

Una vez implementada esta componente, se puede desarrollar el *canvas*, que permite crear **espacios jugables** haciendo click izquierdo en un punto, arrastrando el ratón y soltando el click izquierdo en otro punto. De esta forma, se describen los dos vértices opuestos de un rectángulo, que se crea ajustado al *grid*. Una forma básica como un rectángulo puede observarse en la figura 5.1. Dos espacios que se solapen o compartan un lado se unen formando un polígono más complejo. Además, se permite al jugador eliminar partes del **espacio jugable** haciendo la misma operación que para crear pero con el click derecho del ratón en lugar del izquierdo. De esta forma, se crean espacios complejos como los que se muestran en la figura 5.3

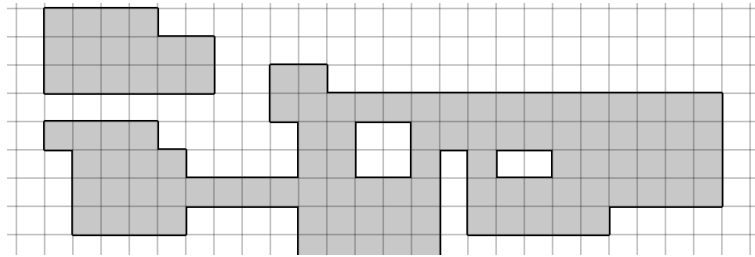


Figura 5.3: Ilustración del tipo de figuras complejas que se pueden diseñar.

5.2.2. Llaves, habitaciones y persistencia

Para el desarrollo del módulo de llaves se ha comenzado definiendo una arquitectura **modelo-vista-controlador**. El modelo representa las llaves, que serán en sí mismos modelos que podrán instanciarse más adelante. La vista es la interfaz gráfica (ver figura 5.4) con la que el usuario puede crear, eliminar y modificar modelos de llaves gracias al controlador, que es una componente lógica.

create key remove key	
key name	is power up
smallKey	<input type="checkbox"/>
doubleJump	<input checked="" type="checkbox"/>
tripleJump	<input checked="" type="checkbox"/>
bigKey	<input type="checkbox"/>

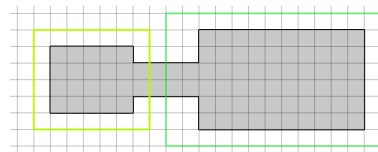
Figura 5.4: Interfaz utilizada para creación, eliminación y edición de los modelos de llaves.

Después de esto, se ha desarrollado un módulo similar para los **niveles**. En éste la vista está compuesta por una componente de lista como se puede observar en la figura 5.5(a) y una componente que forma parte de un *canvas* como se ilustra en la figura 5.5(b). La componente de lista sirve para editar propiedades de los **niveles**, como es el nombre y si es un nivel que será exportado o no. Mientras que la componente del *canvas* sirve para la declaración de qué regiones del diseño conforman los **niveles**. Tal y como se ilustra en la figura 5.5, se pueden declarar niveles rectangulares que no intersequen de igual forma que en el *canvas* se crean los **espacios jugables** (descrito en la subsección 5.2.1). También se puede eliminar uno de estos **niveles** haciendo click derecho dentro de él. Finalmente se puede seleccionar uno con click izquierdo, quedando resaltado tanto en el *canvas* como en la tabla.

Finalmente, se ha integrado un *sub-canvas* que permite colocar instancias de los modelos de las llaves en el diseño, así como instanciar puertas y un token que simboliza la posición inicial del jugador. Junto a esto, se ha implementado una interfaz de texto básica en la consola inferior de la aplicación que muestra la información de las instancias seleccionadas. Todo esto se puede ver reflejado en las figuras 5.6 y 5.7.

Room name	export
start_room	<input checked="" type="checkbox"/>
other_room	<input type="checkbox"/>

(a) En esta interfaz, el usuario puede editar los datos de un nivel ya creado.

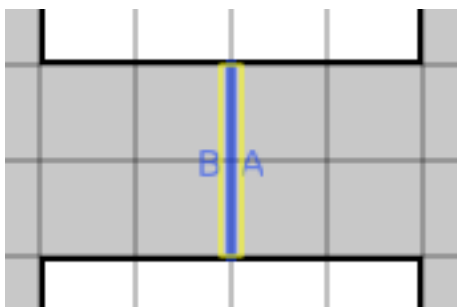


(b) En esta interfaz el usuario puede crear, eliminar y seleccionar niveles. Los niveles declarados son los rectángulos de color verde. El nivel seleccionado es el rectángulo resaltado en color amarillo.

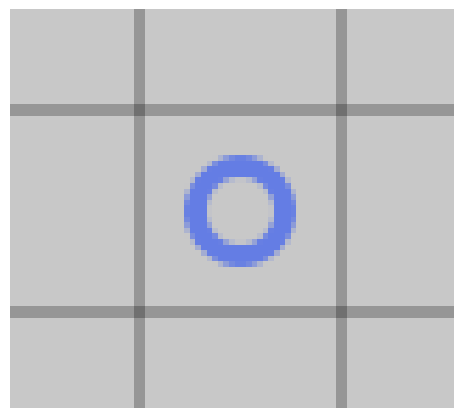
Figura 5.5: la interfaz para interactuar con el controlador de niveles se compone de dos partes.



Figura 5.6: Se permite que el usuario instancie modelos de llaves. Como se puede observar, cuando se selecciona una llave se resalta con un rectángulo amarillo.



(a) El usuario puede crear puertas en el diseño, representadas con líneas azules cuyos lados están identificados con las letras A y B.

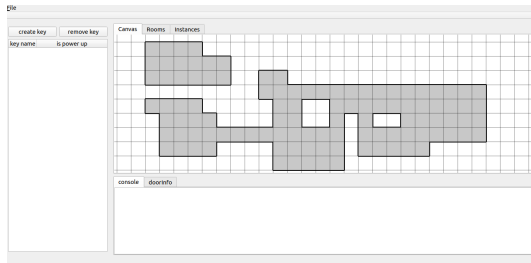


(b) Se puede colocar el token de comienzo, representado con un círculo azul.

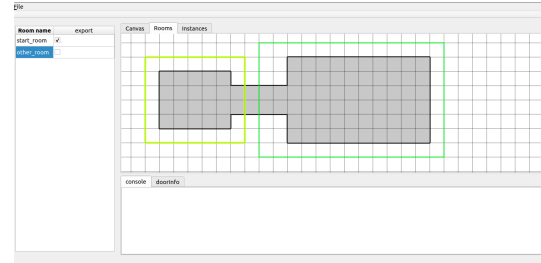
Figura 5.7: Se permite que el usuario instancie puertas y el token de comienzo para la posterior exploración.

5.2.3. Interfaz gráfica

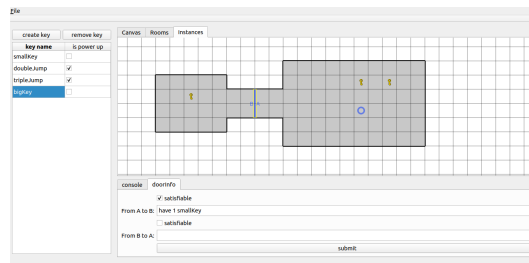
En la fase de desarrollo de la interfaz gráfica de usuario se han integrado todas las componentes en una única interfaz. Resultando en un aspecto como el que se observa en las figuras 5.8 y 5.9.



(a) Esta es la interfaz en la que se inicia el programa, con la pestaña de edición de **espacios jugables** la lista de llaves y la consola auxiliar en la parte inferior.



(b) Esta es la interfaz que resulta al entrar en la pestaña de declaración de **niveles**. Además de mostrar en el *canvas* los declarados, el menú lateral pasa a mostrar también información de los mismos.



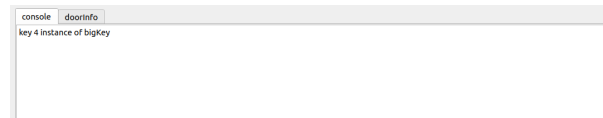
(c) Esta es la interfaz que resulta al entrar en la pestaña de instanciación, en la que se muestran todas las instancias de llaves y puertas creadas, así como el token de comienzo colocado.

Figura 5.8: Capturas diversas de la interfaz, mostrando distintas pestañas.

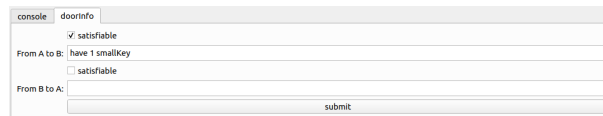
Además de esto, se han desarrollado las funcionalidades de persistencia y de exportación a un proyecto de *GameMaker Studio 2*. Estas funcionalidades se han hecho accesibles a través de la barra de menú superior, así como mediante una serie de atajos de teclado, tal y como se ilustra en la figura 5.10.

La persistencia se lleva a cabo guardando y leyendo ficheros con extensión *.maze* cuyo contenido es un documento *JSON*.

En cuanto a la exportación, se exportan los **niveles** declarados a ficheros de extensión *.yy* compatibles con la estructura de directorios y de proyecto de *GameMaker Studio 2*, de manera que, como se muestra en la figura 5.11, el motor puede abrir los **niveles** creados con la herramienta *MazeDesigner*. Además de crear o editar los ficheros *.yy* que contienen los **niveles**, se ha de modificar también el fichero raíz del proyecto de *GameMaker Studio 2* para incluir estas nuevas habitaciones como recursos nuevos, así como crear otros ficheros *.yy* conteniendo otros recursos necesarios para la exportación (tales como objetos de bloques sólidos, que componen las paredes de los **niveles** exportados).



- (a) Esta pestaña es una consola auxiliar con un campo de texto de solo lectura en el que se muestra información sobre las instancias seleccionadas y otros mensajes enviados por la aplicación.



- (b) Esta pestaña es un formulario para declarar las condiciones que se asignan a una puerta seleccionada.

Figura 5.9: En la parte inferior de la interfaz se muestra una consola auxiliar y una interfaz para la declaración de las condiciones de las puertas.

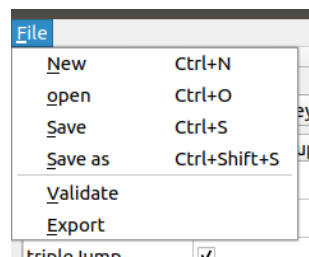
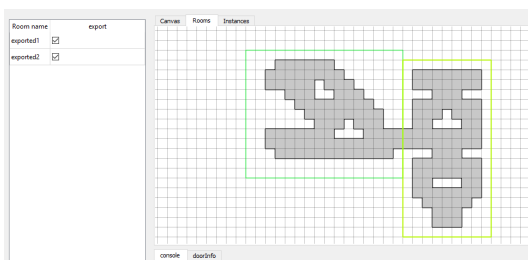
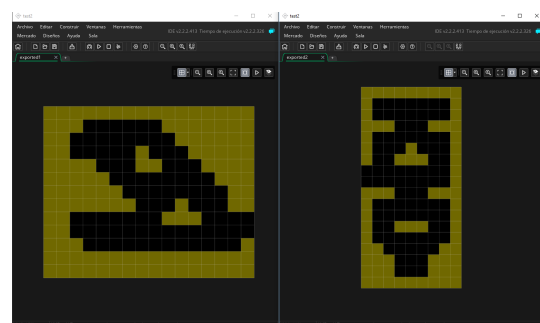


Figura 5.10: Interfaz que muestra una lista de acciones que puede realizar la aplicación, junto con sus atajos de teclado si es que la acción en cuestión tiene alguno asociado.



- (a) El diseño mostrando de los niveles declarados.



- (b) Los niveles declarados en la sub-figura 5.11(a) vistos desde el motor *GameMaker Studio 2*. En el caso del motor, se ha escogido el color amarillo para representar los sólidos, que como se puede ver, coinciden con las regiones no jugables de los niveles declarados.

Figura 5.11: En la parte inferior de la interfaz se muestra una consola auxiliar y una interfaz para la declaración de las condiciones de las puertas.

5.2.4. Condiciones y exploración del diseño

Durante esta etapa se han desarrollado los módulos de condiciones y de exploración del diseño.

El módulo de condiciones tiene como objetivo leer una cadena de caracteres y, si sigue la sintaxis definida en el apéndice A, entonces se creará un objeto de tipo condición. Este objeto se utilizará más adelante en el módulo de exploración. Las condiciones pueden ser: no satisfacibles, la condición vacía (que siempre es satisfacible), condiciones simples (formadas por una única expresión condicional) o una condición compuesta (formada por varias condiciones simples concatenadas por los operadores *or* y *and*).

En cuanto al módulo de la exploración del diseño, se debe formalizar el problema para poder realizar una búsqueda informada hasta alcanzar alguna solución.

El primer paso es convertir los **espacios jugables** del diseño en un grafo equivalente, de manera que dicho grafo representa el diseño lógico implícito que hay en el diseño. Una representación de esta operación se ilustra en la figura 5.12.

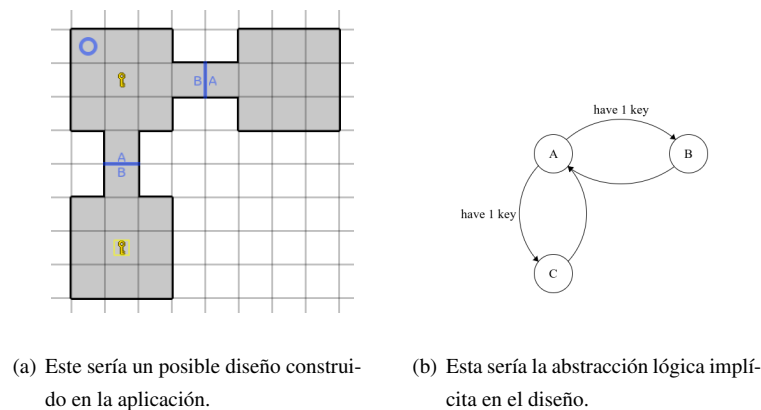


Figura 5.12: En esta figura se muestra como sería la conversión de un diseño a un grafo, si las condiciones en ambas puertas fueran la vacía en una dirección y tener una llave en la otra.

El **grafo dirigido** generado por el diseño está formado por una serie de nodos y aristas.

Los nodos son construidos a partir de las regiones conexas del diseño, que contienen referencias a las llaves que estuvieran contenidas en dichas regiones.

Las aristas se construyen por pares a partir de las puertas, conectan dos nodos de forma unidireccional y poseen una condición.

Una vez se ha logrado dicha transformación se formaliza el problema de la siguiente forma:

- Se dice que un nodo **A** es **vecino de un nodo B** si existe una arista que une B con A en esa misma dirección.

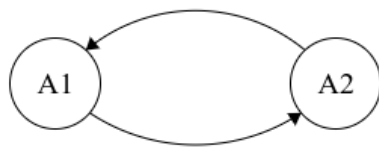
- La **relación de vecindad no es simétrica**, es decir, no es suficiente que A sea vecino de B para afirmar que B es vecino de A.
- Se define un **grafo del diseño** como el **grafo dirigido** resultante de la abstracción de un diseño a grafo.
- Se define un **estado de búsqueda** como un grafo del diseño, que posee una serie de nodos, aristas que conectan dichos nodos, un nodo actual en el que se encontraría el jugador y un inventario con todos los objetos que ha recolectado.
- Se define una **acción** como una modificación de un estado de búsqueda, ya sea moviendo el nodo actual a alguno de sus vecinos o extrayendo un objeto del nodo actual para almacenarlo en el inventario.
- Se define el **coste de viaje de una arista** como la lista de los objetos consumibles que han de emplearse para satisfacer la condición que dicha arista posee.
- Se define la **operación de viajar** desde un nodo A hasta un nodo B como la acción de mover el nodo actual de un estado desde A hasta B a través de una arista, asumiendo el coste de viaje de la misma.
- Se define **exploración** como una sucesión de acciones.
- Se dice que la **puerta que une A con B está abierta** en esa dirección si se ha realizado una operación de viajar desde A hasta B y, en ese caso, el coste de la arista que une A con B pasa a ser la lista vacía.
- Se dice que el conjunto de nodos A_1, A_2, \dots, A_n es **fuertemente conexo** [11] si existen puertas abiertas que unen A_i con A_{i+1} para todo i desde 1 hasta $n-1$ y existe una puerta que une A_n con A_1 que también está abierta. En nuestra abstracción del problema, hemos utilizado la definición de nodos fuertemente conexos tradicional [11] únicamente considerando el subgrafo resultante de ignorar toda puerta no abierta.
- Dado un conjunto de nodos A_1, A_2, \dots, A_n fuertemente conexo, se define la operación de **unión de nodos** como aquella que convierte el conjunto de esos n nodos en un único nodo B. todas las aristas que unían cada uno de los nodos A_i (con i desde 1 hasta n) con otros nodos (que no pertenecen al conjunto de los A_i) ahora unen B con dichos nodos. Las aristas que unían A_i con A_{i+1} desaparecen, pero toda arista que unía A_j con A_k (para $j \neq k + 1$ y $k \neq j + 1$ por ser puertas abiertas) tal que el coste no es la lista vacía, se conservan, conectando ahora B con el propio B (ver figura 5.13).
- Se dice que una exploración es **correcta** si la serie de acciones que la define concluye convirtiendo el grafo en un único nodo.
- Se dice que una exploración es **incorrecta** si la serie de acciones que la define concluye en un estado de búsqueda en el que no hay más acciones disponibles.
- Se dice que un diseño es **correcto** si todas las posibles exploraciones son correctas. De lo contrario, decimos que un diseño es **incorrecto**.

Para determinar que un conjunto de nodos es fuertemente conexo se ha implementado el algoritmo de Tarjan [11] que, dado un grafo dirigido, encuentra sus componentes fuertemente conexas. Siguiendo este algoritmo, además, el resultado es una partición maximal del grafo (bajo la relación de equivalencia de conexión fuerte) en la cual cada nodo solamente puede pertenecer a una de estas componentes.

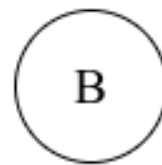
Con estas formalizaciones, se puede realizar una búsqueda **A-estrella** [12] [13] para determinar si el diseño creado por el usuario es correcto o no.

Nótese que el token de comienzo colocado según se expone en la subsección 5.2.2 es imprescindible para la exploración, pues al comienzo de la misma se ha de determinar cual empieza siendo el nodo actual.

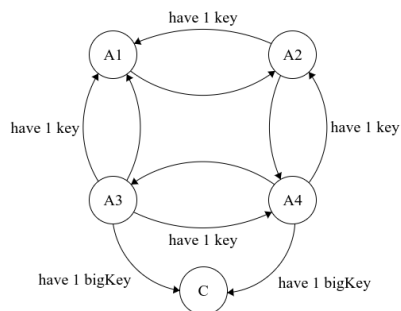
Dado que demostrar que el diseño es correcto siempre tiene la complejidad máxima de la explora-



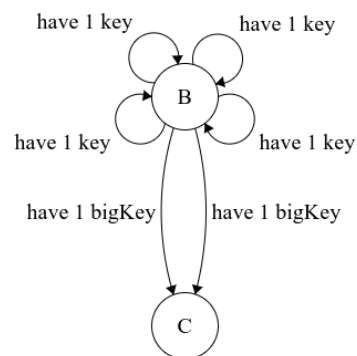
(a) El caso básico en el que dos nodos están conectados por aristas con la condición vacía.



(b) el caso 5.13(a) se reduce a este grafo tras la operación de unión.



(c) Un caso más complejo de nodos fuertemente conexos.



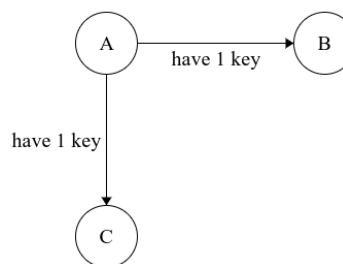
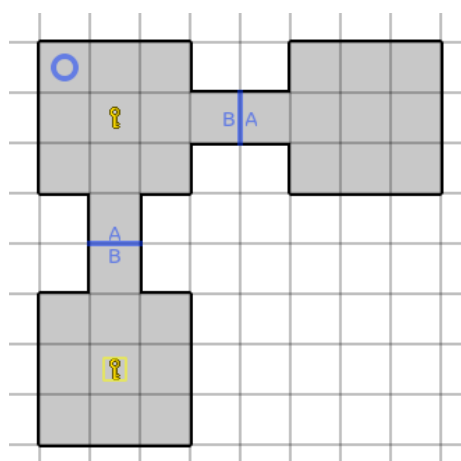
(d) el caso 5.13(c) se reduce a este grafo tras la operación de unión.

Figura 5.13: En esta figura se ilustra la operación de unión de nodos. Las sub-figuras 5.13(a) y 5.13(b) muestran un ejemplo básico mientras que las sub-figuras 5.13(c) y 5.13(d) muestran un ejemplo más completo

ción, interesa desarrollar una función **heurística** que permita encontrar exploraciones incorrectas, si las hubiera, tan pronto como fuera posible. Es por eso que se escoge una función **heurística** “derrochadora”, que evalúe mejor estados en los que el jugador tiene pocos objetos en su inventario y en los que los nodos accesibles posean también pocas llaves, ya que esto provoca que tenga menos opciones para explorar y se maximizan las probabilidades de quedar atrapado o de dejar inaccesible algún área del diseño.

De esto se extrae una condición necesaria pero no suficiente para que el diseño sea correcto: para dos nodos A y B cualquiera del grafo del diseño, debe existir un camino que, ignorando las condiciones de las aristas, una A con B y viceversa.

Un ejemplo de un diseño incorrecto debido a no cumplir esta condición sería el mostrado en la figura 5.14.



(a) Esta sub-figura muestra el diseño del que se hace la abstracción lógica implícita. Nótese que a priori es el mismo diseño que el expuesto en la figura 5.12 a excepción de las condiciones que se asignan a las puertas.

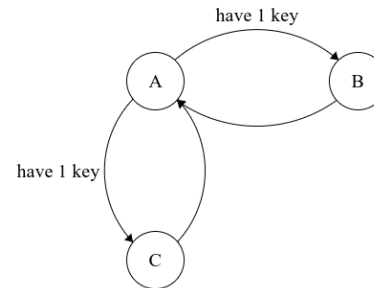
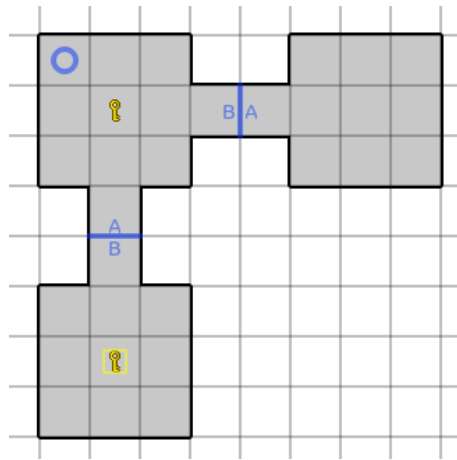
(b) Esta sub-figura muestra la abstracción lógica implícita en el diseño de la sub-figura 5.14(a). Como se puede observar, no existe ningún camino que una el nodo B con el nodo A ni tampoco existe ninguno que una el nodo C con el nodo A. Bastaría con que el jugador llegara a dicho nodo para que su progreso quedara bloqueado.

Figura 5.14: En esta figura se ilustra un ejemplo de diseño que no cumple con la condición necesaria para ser un diseño correcto.

Finalmente, si se encuentra una exploración incorrecta, se indica en el *canvas*, mediante un camino en rojo, las acciones que podría tomar un jugador para quedar atrapado o dejar un área inaccesible. Tal y como se ilustra en la figura 5.15.

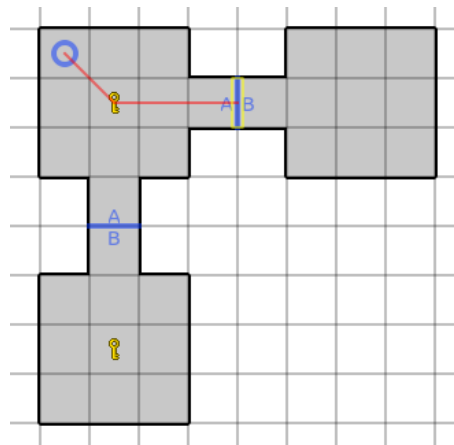
5.3. Dificultades encontradas

Durante el desarrollo de la aplicación, se han detectado dificultades notables. Tales como:



(a) Este sería el diseño construido por el usuario.

(b) Esta sería la abstracción lógica implícita en el diseño creado por el usuario.



(c) En esta sub-figura se ilustra cómo se muestra al usuario que su diseño es incorrecto y la exploración que sirve como prueba. En este caso el diseño es incorrecto porque, en una exploración, se consume la única llave de la que se dispone para abrir la puerta del este, ya que tras eso, no hay forma de obtener otra llave para abrir la puerta del sur, quedando el área del sur inaccesible permanentemente.

Figura 5.15: En esta figura se ilustra un diseño incorrecto y cómo la aplicación muestra al usuario la exploración que podría realizar el jugador que resulta en que un área (la que se encuentra al sur) queda completamente inaccesible.

- La implementación de la clase `CompositeCondition` que representa las condiciones compuestas resultó ser demasiado compleja y, dado que se pueden replicar las condiciones compuestas con cierto ingenio colocando puertas en el diseño, se ha optado por dejarla para trabajos futuros.
- La exploración que se ha de realizar para resolver el problema planteado tiene una alta complejidad, de manera que se requieren optimizaciones futuras del código para mitigar este tiempo de computación. A pesar de lo cual, para diseños suficientemente grandes, la carga computacional crecerá muy rápidamente, de manera que será inevitable que la validación de dichos diseños tome tiempo.
- Durante el desarrollo se han encontrado algunos bugs relacionados con las interacciones de los módulos. Se han resuelto todos los posibles, pero es posible que haya aun más bugs por detectar.
- La implementación de algunos *sub-canvas* dependían de otros módulos, de manera que hay cierto acoplamiento entre algunos de ellos.

INTEGRACIÓN, PRUEBAS Y RESULTADOS

Para comprobar el correcto funcionamiento de los módulos se han llevado a cabo una serie de pruebas, tanto a nivel individual como a nivel colectivo, con el objetivo adicional de asegurar el cumplimiento de los requisitos establecidos para el producto software.

6.1. Pruebas unitarias

Con cada módulo, se han realizado pruebas unitarias que aseguran la correcta ejecución del código, no solo a nivel íntegro de la aplicación, en tanto a que no haya errores técnicos que cierren el programa, sino también a nivel de usuario, en tanto a que cada módulo hace las operaciones que se esperan de él y cómo se esperan de él.

Para el módulo del Canvas, se han realizado pruebas de añadir y eliminar espacios tanto del interior como de los bordes del diseño, a distintas ampliaciones del Grid y habiéndolo trasladado tanto hacia coordenadas negativas como a positivas. También se ha intentado eliminar espacios donde no había antes, se ha intentado añadir y eliminar espacios con área cero (puntos y líneas) y se ha comprobado que el funcionamiento de la creación y eliminación de espacios es el esperado. Con respecto a las adiciones siguientes a este módulo, se ha comprobado que no se pueden definir habitaciones con intersección no vacía y que se eliminan como cabe esperar. la creación de las mismas funciona también correctamente tal y como sucede con el Canvas. A la hora de instanciar llaves, puertas y el token de comienzo, se ha comprobado que solamente se pueden instanciar en el interior de un **espacio jugable**, que las instancias creadas pueden ser efectivamente eliminadas y que no se pueden colocar dos instancias en la misma posición.

En el módulo de las llaves, se ha comprobado que se pueden crear y eliminar modelos de llaves correctamente y que estos cambios en la interfaz se ven reflejados en los modelos que hay en memoria. Se ha probado a eliminar una llave sin seleccionar ninguna, a eliminar varias a la vez, a crear llaves con el mismo nombre y a realizar cambios sobre ellas.

En el módulo de condiciones se ha comprobado que solamente se admiten condiciones que cumplan con la sintaxis definida en el apéndice A y que estas pueden ser validadas y pueden comprobar

si son satisfacibles. También se ha comprobado que devuelven correctamente la lista con los objetos a consumir en caso de poder ser satisfacibles.

En el módulo de persistencia se ha comprobado que los datos guardados y esos mismos datos posteriormente cargados de fichero coinciden.

En la exportación se ha comprobado que se generan ficheros .yy de acuerdo a la estructura de proyecto de *Game Maker Studio 2*, creándolos cuando el proyecto en cuestión no los tiene y editándolos en caso contrario, sin sobrescribir ningún dato innecesariamente del proyecto.

En el módulo de exploración se ha comprobado con diseños incorrectos que el programa puede detectar los fallos y devuelve la secuencia de acciones a tomar para replicar el problema. También se ha comprobado que el programa detecta cuando un diseño es correcto.

6.2. Pruebas de integración

La integración de los módulos se ha realizado de forma iterativa en el desarrollo del proyecto. Con cada integración se han realizado pruebas para asegurar que el sistema funciona en conjunto. La mayoría de estas pruebas han consistido en comprobar la correcta interacción entre distintos componentes de la aplicación.

Se ha comprobado que seleccionando elementos del Canvas, se cargan los detalles de los mismos en la consola de la aplicación. Que al cambiar al contexto de la definición de las habitaciones, se muestra la lista de habitaciones con sus datos en lugar de la lista de los modelos de llaves y que en la pestaña de edición de las condiciones de las puertas efectivamente se pueden definir las condiciones y son aplicadas a la puerta seleccionada. También se ha comprobado que la eliminación de un modelo de llave supone la eliminación de todas sus instancias en el Canvas y que, en una condición que requiere un modelo de llave inexistente, se reporta un error y la condición se convierte en insatisfacible.

Además, al eliminar espacios jugables que contenían alguna instancia de llave, éstas son eliminadas para asegurar que no pueden existir fuera de algún espacio jugable. La misma comprobación se ha hecho para el token de comienzo y para las puertas.

6.3. Resultados

El resultado tras el periodo de desarrollo es un producto software que cumple con la mayoría de los requisitos establecidos, dejando los pocos que no se cumplían para trabajo futuro.

El producto software desarrollado está preparado para ser utilizado en el desarrollo de videojuegos.

CONCLUSIONES Y TRABAJO FUTURO

Una vez finalizadas las etapas del desarrollo, se ha construido un producto software funcional que cumple con los requisitos. Sin embargo, el ciclo de vida del desarrollo no concluye. En esta sección se habla del producto logrado, la utilidad que puede tener y apartados del mismo que quedan para ser ampliados, añadidos o mejorados en el futuro.

7.1. Conclusiones

Maze Designer es un producto software que cumple con los requisitos establecidos para ser utilizada como una herramienta de diseño de niveles en juegos de tipo mazmorra tal y como se ha explicado a lo largo de esta memoria.

Cubre un área de trabajo bastante específica en la industria del videojuego pero sin embargo, el uso de esta herramienta facilita mucho tareas que no son triviales en el diseño de los niveles. A la vez, estas facilidades pueden dar lugar a un nuevo estilo de diseño de los niveles de un juego de tipo mazmorra, permitiendo que se exploren las posibilidades de un diseño orientado a una **exploración ramificada**. Además, al estar preparada para exportar a *Game Maker Studio 2*, puede ser incorporada en el **pipeline** del desarrollo sin mayor esfuerzo por parte del equipo de desarrollo. Aunque esto no significa que los diseños elaborados con esta herramienta sean específicos para *GameMaker Studio 2*.

Así mismo, *Maze Designer* es una herramienta que puede ser utilizada para experimentar con patrones de diseño de forma poco costosa. Acelerando el aprendizaje de nuevos diseñadores de niveles.

7.2. Trabajo futuro

Con la finalización del desarrollo, se plantean las siguientes tareas que podrían realizarse en el futuro para mejorar o ampliar las funcionalidades de *Maze Designer*:

- 1.– **Implementación de condiciones compuestas:** las condiciones actualmente soportadas por la aplicación son condiciones simples. Se deja para el futuro la implementación de condiciones compuestas que permitirían mayor flexibilidad en los diseños.
- 2.– **Exportar a otros motores:** se ha desarrollado la aplicación con *Game Maker Studio 2* como motor de videojuegos al que exportar, sin embargo, debido a que los diseños son genéricos, se podría explorar la posibilidad de exportarlos a otros motores, abarcando un mayor número de usuarios.
- 3.– **Incluir un sistema de notas:** a la hora de desarrollar esta *Maze Designer*, no se tuvo en cuenta una funcionalidad que podría ser útil, como es la de poder apuntar notas en cualquier parte del diseño.
- 4.– **Implementar otros patrones:** en esta aplicación se ha prestado especial atención a dos de los patrones de diseño predominantes en los juegos de tipo mazmorra, sin embargo, existen otros patrones que no se han tenido en cuenta como *triggers* o *warps*. Éstos podrían ser una adición interesante para la aplicación.
- 5.– **Mantenimiento:** como todo proyecto software, es inevitable la presencia de bugs en un código. Se delega al trabajo futuro la corrección de estos errores según sean reportados.
- 6.– **Realización de pruebas por parte de usuarios:** en el desarrollo de este TFG no se han podido realizar pruebas de la aplicación por parte de usuarios reales más allá del desarrollador. Convendría realizar una serie de pruebas para el control de la calidad del producto, de manera que se obtuviera retroalimentación para poder seguir mejorando la aplicación.
- 7.– **Mejorar el rendimiento de la aplicación:** debido al tiempo de desarrollo, no se han podido optimizar todos los módulos al máximo, utilizando el potencial que ofrece C++. Una optimización de la herramienta podría significar una mejor experiencia del usuario y, en particular, en las tareas de exploración, podría reducirse el tiempo de cómputo.

BIBLIOGRAFÍA

- [1] A. Ezust and P. Ezust, *An introduction to design patterns in C++ with Qt 4*. Prentice-Hall, 2007.
- [2] J. Blanchette and M. Summerfield, *C++ GUI programming with Qt 4*. Prentice Hall Professional, 2006.
- [3] O. Denninger and J. Schimmel, “Game programming and xna in software engineering education,” *Proceedings of Computer Games and Allied Technology (CGAT08)*, 2008.
- [4] J. L. Ford Jr, *Getting Started with Game Maker*. Cengage Learning, 2009.
- [5] “General information, tutorials, references and articles about the c++ programming language.” <http://www.cplusplus.com/>.
- [6] “w3schools about c++.” https://www.w3schools.com/cpp/cpp_intro.asp.
- [7] “official qt documentation.” <https://doc.qt.io/>.
- [8] S. Huang, *Qt 5 Blueprints*. Packt Publishing Ltd, 2015.
- [9] D. Spinellis, “Git,” *IEEE software*, vol. 29, no. 3, pp. 100–101, 2012.
- [10] T. Bray, “The javascript object notation (json) data interchange format,” tech. rep., 2017.
- [11] E. Nuutila and E. Soisalon-Soininen, “On finding the strongly connected components in a directed graph,” *Information Processing Letters*, vol. 49, no. 1, pp. 9 – 14, 1994.
- [12] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.
- [13] N. J. Nilsson and N. J. Nilsson, *Artificial intelligence: a new synthesis*. Morgan Kaufmann, 1998.

DEFINICIONES

A-estrella algoritmo de búsqueda que encuentra el camino de menor coste entre un estado y otro dentro del espacio de búsqueda. El coste de los caminos se define según una heurística.

backtracking En una exploración, la acción de regresar a localizaciones anteriormente exploradas.

dead lock Estado de la exploración del grafo de un **diseño** a partir del cual existen nodos que dejan de ser accesibles aunque se continuara con dicha exploración.

diseño Conjunto de **espacios jugables**, no necesariamente conexos, que representa la totalidad de los espacios virtuales de juego.

editor de niveles Herramienta utilizada por los motores de videojuegos para colocar las instancias que constituyen un escenario del juego.

espacio de juego ver **espacios jugables**.

espacio jugable regiones o terreno virtuales de un juego por el cual el jugador podrá mover a su avatar.

exploración lineal Tipo de exploración caracterizada por carecer de múltiples caminos por los que progresar.

exploración ramificada Tipo de exploración caracterizada por presentar múltiples caminos por los que progresar.

grafo estructura abstracta representada por un conjunto de nodos y un conjunto de conexiones entre ellos. A estas conexiones se las denomina aristas.

grafo dirigido categoría de **grafos** en los cuales, la relación de conexión entre nodos no es recíproca. Es decir, un nodo A puede estar conectado con un nodo B pero no necesariamente se cumple que dicho nodo B esté conectado al nodo A, porque la dirección de la arista que los conecta es desde A hasta B.

heurística función escalar que recibe un estado del espacio de búsqueda y devuelve un número mayor que cero.

mazmorra Conjunto de **niveles** cuya exploración y cuyas conexiones están bloqueadas o limitadas por una serie de requisitos que se han de cumplir para progresar.

modelo-vista-controlador arquitectura formada por tres componentes que interactúan entre sí. El modelo representa la estructura de los datos, la vista la interfaz con la cual se accede al controlador, que se encarga de modificar datos que siguen la estructura del modelo.

nivel subconjunto del total de los **espacios jugables** en los que se desarrolla la acción. Suele tratarse de un área reducida del total de ellos, representando una (o parte de una) localización o **pantalla**.

objeto consumible Categoría de objetos del videojuego que tras ser utilizados desaparecen.

orientación a objetos Paradigma de programación imperativo que utiliza objetos pertenecientes a clases y sus interacciones.

pantalla Regiones del **diseño** que encapsulan los **espacios jugables**. Un juego en tiempo de ejecución muestra solamente una de estas regiones.

pipeline Conjunto de fases que componen el desarrollo de un producto software. Dichas fases se disponen de manera secuencial, estableciendo una cadena de dependencias.

power-up Categoría de objetos del videojuego que tras ser recolectados el jugador puede utilizarlos tantas veces como quiera.

programación estructurada Paradigma de programación imperativo cuya abstracción se limita al uso de subrutinas.

programación funcional Paradigma de programación declarativo que se basa en modelar el problema con una serie de funciones matemáticas.

programación genérica Estilo de programación que se centra en el funcionamiento de los algoritmos aplicados a objetos a especificar en tiempo de ejecución.

trigger patrón de diseño que modela una consecuencia arbitraria de la realización de alguna acción, ya sea atravesar un cierto área u obtener un cierto objeto.

videojuego de tipo mazmorra categoría de producto de ocio digital cuyo entretenimiento gira en torno a la exploración de una serie de localizaciones para progresar.

warp patrón de diseño que modela la conexión de dos espacios distantes a través de una mecánica como el teletransporte.

APÉNDICES

DEFINICIÓN DE LA SINTAXIS DE

CONDICIONES

Código A.1: a definición de la sintaxis seguida para las condiciones, escrita en el lenguaje prolog. Nótese que la cláusula *item* ->[*key*] debería admitir cualquier cadena de caracteres alfanuméricos que no contenga espacios, pero se deja así por simplificar el código debido a que esta especificación es puramente formal y no se emplea en el desarrollo como tal.

```
1  startCondition --> condition.
2  startCondition --> [].
3  condition --> simpleCondition.
4  condition --> compositeCondition.
5  simpleCondition --> have, number, item.
6  have --> [have].
7  have --> [].
8  number --> numberConstant.
9  number --> [].
10 numberConstant --> nonZeroDigit.
11 numberConstant --> nonZeroDigit, digits.
12 nonZeroDigit --> [1].
13 nonZeroDigit --> [2].
14 nonZeroDigit --> [3].
15 nonZeroDigit --> [4].
16 nonZeroDigit --> [5].
17 nonZeroDigit --> [6].
18 nonZeroDigit --> [7].
19 nonZeroDigit --> [8].
20 nonZeroDigit --> [9].
21 digits --> digit.
22 digits --> digit, digits.
23 digit --> nonZeroDigit.
24 digit --> [0].
25 item --> [key].
26 compositeCondition --> conditionList.
27 compositeCondition --> conditionList, connector, compositeCondition.
28 compositeCondition --> ["("], conditionList, [")"], connector, compositeCondition.
29 compositeCondition --> ["("], compositeCondition, [")"].
30 conditionList --> simpleCondition.
31 conditionList --> simpleCondition, connector, conditionList.
32 connector --> [or].
33 connector --> [and].
```


PATRÓN DE DISEÑO DE NIVELES

EMPLEADO COMO TUTORIAL

En el diseño de juegos de tipo mazmorra es común utilizar un patrón de diseño de niveles unidireccional para enseñar al jugador, sin que este lo note, cómo funciona una mecánica de juego recién introducida. Para ilustrar cómo se realiza esto, se emplea un ejemplo clásico, que ha sido imitado a lo largo de los años en múltiples videojuegos. Dicho ejemplo se explica en la figura B.1.



Figura B.1: Escenario del videojuego *Metroid*¹ (Nintendo, 1986). La imagen ha sido editada para realizar una enumeración en base a la cual se explica el patrón relativo a este apéndice. En este escenario, el jugador llega desde la parte derecha de la pantalla, encontrándose con la estructura sólida que vemos entre 2 y 4. Dado que 4 es un túnel demasiado pequeño para que el jugador quepa, accederá a este área subiendo por 1 yasta alcanzar 2 y descender hacia 3. En este punto, el jugador queda atrapado, pues su salto no es suficientemente alto como para alcanzar 2 desde 3. Afortunadamente, en 3 hay un *power-up* denominado *morph ball* que, en ese juego, permite al jugador pasar por túneles estrechos como 4. Así pues, la única forma que el jugador tiene para salir de ahí y continuar con el juego, es utilizar el objeto que acaba de adquirir, aprendiendo la nueva mecánica de juego introducida por dicho *power-up*.

Como se puede estudiar de este ejemplo, se ha utilizado una transición unidireccional para dejar atrapado al jugador en una zona determinada. La única forma que tiene para escapar de dicha zona es mediante el uso de una mecánica nueva, de manera que los diseñadores se aseguran de que el jugador aprende cómo funciona esta mecánica sin que el jugador se de cuenta de que le están enseñando.

La abstracción lógica de este patrón se ilustra en la figura B.2.

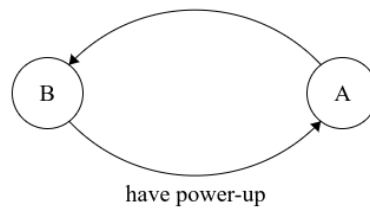


Figura B.2: Abstracción lógica del patrón tratado en este apéndice. En este esquema se asume que el nodo B contiene el power-up necesario para viajar desde B hasta A.